Exercises – Summary

Objective

To describe the exercises in the course.

Overview

The instructor will have shown you the install location of your courseware folder.

Folders

Within your courseware installation folder, you should have an **Exercises** folder.

Within the Exercises folder, exercises for each of the chapters are provided.

Each exercise folder has a:

- Starter folder – Where you will create, edit and run scripts
- Solution folder – Contains finished examples for you to compare with your work

Every exercise is independent of the previous one. That means you don't need to finish one exercise to be able to start the next one.

If you have time sections

Most exercises have **if you have time** questions that you can work on if you have enough time. They are intended to cater for delegates who have joined the course with existing programming experience and are intended to challenge. Don't worry if you don't manage to complete these sections.

Working at Home

To work on the exercises at home you will need to download and install Python 3.7 or later from http://www.python.org\downloads.

Integrated Development Environment

We recommend you download and install PyCharm Community Edition from https://www.jetbrains.com/pycharm/download.

Exercise 1 – Understanding computers

Objective

To understand how computers work under the hood.

Duration

About 20-30 minutes.

Overview

CPUs understand their own language, a language written in binary. Every action a CPU can take has a specific instruction with a known binary value. For example, the ADD instruction might be understood by the CPU as 0100101.

In this exercise, you are going to come up with your own instruction set for a programmable device that can move in space rather perform data operations.

This device can draw on paper. It can Move Up, Down, Left, Right and can put its pen up or down to draw when it needs to. Moving with the pen down draws on the paper.

Step 1 – Define your instruction set

Define an instruction set that includes actions for moving in different directions, engaging/dis-engaging the pen with the paper and maybe even changing pen colours.

Each instruction should have a unique decimal number associated with it.

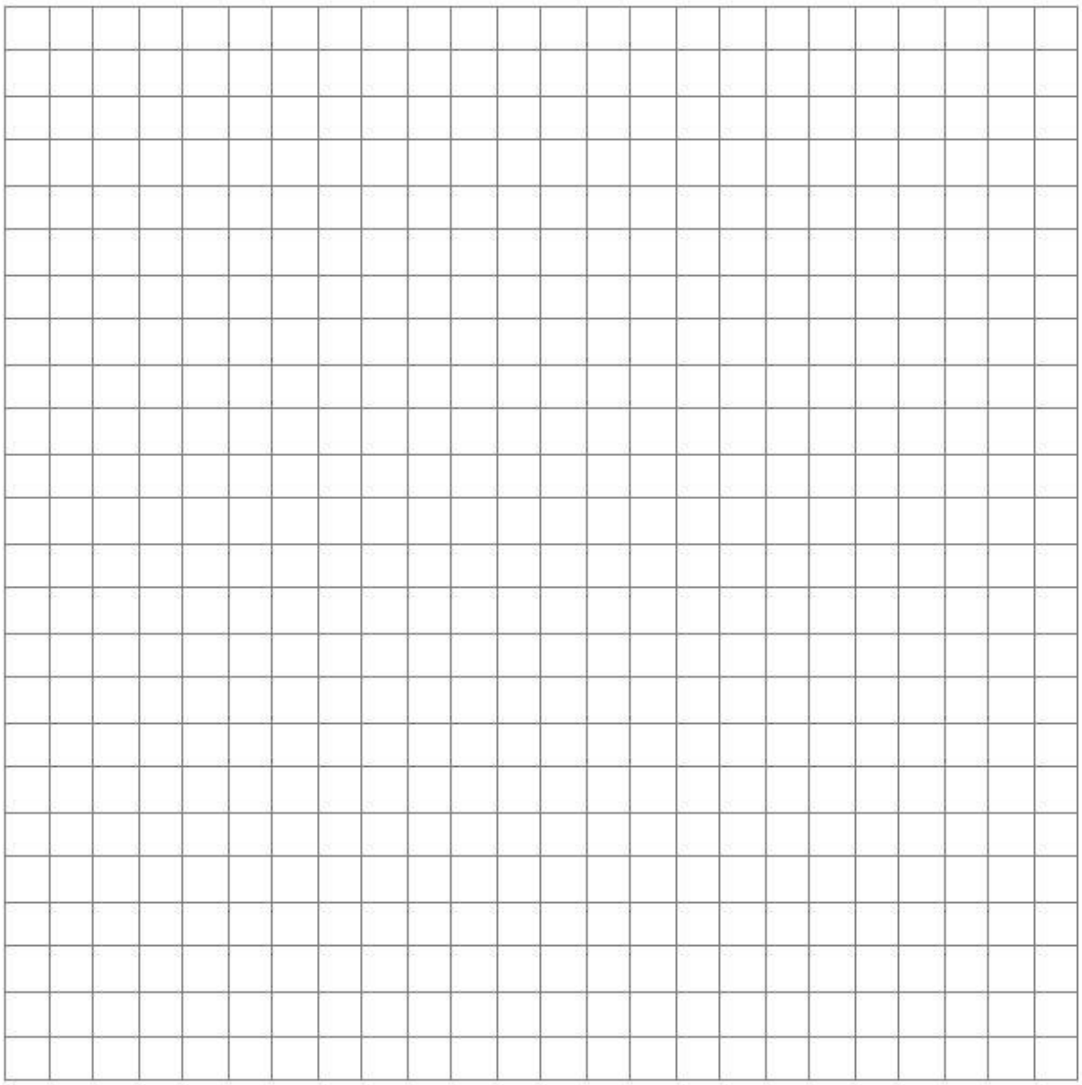
Example:

Source Commands	Native Command
MOVE_UP	25

Step 2 – Write a program

Using the Source Commands, you defined above, write a program that would draw the number 2 on a grid similar to below with the origin at the top left. One box is one point.

MOVE_UP 1 would move up 1 box.



Example

```
MOVE_RIGHT
5
PEN_DOWN
MOVE_RIGHT
5
```

Step 3 – Compile your program

You will now act as a compiler and compile your source into its equivalent native command (in decimal).

E.g. the above might compile down to:

```
25 5 45 25 5
```

These decimal instructions can now be executed directly by the device.

Step 4 – Work as a team

Ask your neighbour or instructor to execute your decimal program. You will need to give them your instruction set.

Step 5 – (if you have time) Compile to binary

Dare you compile to binary? Don't worry if you don't know binary, we will see how binary numbering works in the next chapter.

Exercise 2 – Bits and bytes

Objective

To learn about bits, bytes and CPUs.

Duration

About 15-20 minutes.

Overview

Bits and Bytes are the core of every computer system. They store the data and the program instructions that make everything work.

Part 1 – Data quiz

We will start off with a self-quiz to gently test your knowledge of bits so far. The point of this is to help you identify gaps in your knowledge that you may like to talk through with your instructor.

Question	True	False
1. Computers use transistors to store “bits” of information		
2. A bit consists of eight 1s or 0s		
3. In binary, 1 is the equivalent to “On” and 0 to “Off”		
4. The maximum value you can store in 8 bits is 256		
5. The total number of values you can store (including zero) in 8 bits is 256		
6. RAM stands for Random Access Memory		
7. A CPU’s primary purpose is to store data		
8. The CPU is the brain of the computer		

Part 2 – Converting numbers

Binary uses 1s and 0s to describe much bigger numbers. Using a bit pattern, individual bits can be set to represent precise number values.

128	64	32	16	8	4	2	1
0	0	0	0	0	1	1	1

In the example above, the 1, 2 and 4 bits have been set which means we can add up all of those numbers to give the decimal result **7**

0x8=0

1x4=4

1x2=2

1x1=1

0+4+2+1=7

Question	Answer
1. What would the number 5 be in binary?	
2. Translate 00010001 into decimal?	
3. 129 in binary?	
4. 01000001 in decimal is?	
5. What is the decimal number 15 in hex?	

Answers

Part 1 – Data quiz

Question	True	False
1. Computers use transistors to store “bits” of information	•	
2. A bit consists of eight 1s or 0s		•
3. In binary, 1 is the equivalent to “On” and 0 to “Off”	•	
4. The maximum value you can store in 8 bits is 256		•
5. The total number of values you can store (including zero) in 8 bits is 256	•	
6. RAM stands for Random Access Memory	•	
7. A CPU’s primary purpose is to store data		•
8. The CPU is the brain of the computer	•	

Part 2 – Converting numbers

Question	Answer
1. What would the number 5 be in binary?	101
2. Translate 00010001 into decimal?	17
3. 129 in binary?	10000001
4. 01000001 in decimal	65
5. What is the decimal number 15 in hex?	F

Exercise 3 – Data and Type

Objective

To understand data types and how to work with them.

Duration

15-20 minutes.

Overview

A collection of bits means nothing unless we have some context, some understanding of what those bits are supposed to mean. A collection of 8 bits could equally refer to an alphabetic character or a whole number. To understand how to interpret the bits requires us to know the data item's data type.

Part 1 – Declaring data

Open your Python Integrated Development Environment (PyCharm or IDLE). Your instructor should already have shown you how to do this. Ask her/him to show you if not.

Create a new Python script using (**File > New Python File**) then choose

File > Save As and save your file as Ex3.py in the folder C:\Labs\Exercise 3\Starter directory.

We want you to declare some data variables to store the following information about your favourite movie:

- Title
- Director
- Year of release
- Duration
- Certificate

(Use IMDB.com if you are not sure)

Once you have done that print each of the data variables. You can use more than one print function if necessary.

Example:

If I had stored my name and age in two variables called **name** and **age,** I could print them using the command:

```
print(name, age)
```

You could take this further by adding some annotating text:

```
print("My friend", name, "is", age, "years old")
```

See below for a description of the steps you need to take to achieve this.

To run your script, in PyCharm use SHIFT+F10, in IDLE use F5, or choose from the main menu **Run > Run Module**

Steps

1.	Declare a variable called title and assign the title of your favourite movie
	`title = "The Flintstones"`
2.	Declare a variable for Director
	`director = "Brian Levant"`
3.	Repeat above for Year, Duration and Certificate

	`year = 1994`
	`duration = 91`
	`cert = "PG"`

4.	Print the data
	`print(title, "was directed by", director, "in", year)` `print("It's duration is" , duration, "minutes and it is a", cert, "certificate")`

Part 2 – Capturing data

Next, let's change the code so that instead of hard coding the movie data, we allow the user to enter it.

You will need to use the i**nput** function for this.

An example of **input** to capture a person's name might look like this:

```
first_name = input("What is your first name? ")
```

Have a go at making this change for some or all of: title, director, year, duration and certification. The steps are provided below if you need them.

Steps

5.	Change the line `title = "The Flintstones"`
	To `title = input("Movie Title: ")`
6.	Repeat for director, year, duration and cert

Remember: input returns a string so you may need to convert string to int for some data. Ask your instructor about this if you are not sure.

Part 3 – Determining type (if you have time)

Python in common with many strongly typed languages remembers the type of a variable. To find out what type a variable belongs to you can use the **type** function.

The next step then is to determine the type of all of the variables you declared earlier.

Hint:

If you had a variable **x** to find its type you would write:

```
print(type(x))
```

Answers

Part 1

```
title = "The Flintstones"

director = "Brian Levant"

year = 1994

duration = 91

cert = "PG"

print(title, "was directed by", director, "in", year)

print("It's duration is", duration, "minutes and it is a", cert,
"certificate")
```

Part 2

```
title = input("Movie Title: ")

director = input("Director: ")

year = int(input("Year: "))

duration = int(input("Duration (mins): "))

cert = input("Cert: ")

print(title, "was directed by", director, "in", year)

print("It's duration is", duration, "minutes and it is a", cert,
"certificate")
```

Part 3

```
print(type(title))

print(type(director))

print(type(year))

print(type(duration))

print(type(cert))
```

Exercise 4 – Performing Operations on Data

Objective

In this exercise, we will use operations to calculate new data items from existing values.

Duration

15-20 minutes.

Overview

Often one of a program's main tasks is to perform calculations and perform transformations on data. Understanding how to apply operations to data is vital.

Part 1 – Performing operations

There's always more than one way to do most things. The same is true for operations on data.

Assuming I have a variable defined as **speed = 0,** I could increase the speed using either line below:

```
speed = speed + 10

speed += 10
```

1.	Create a new Python script from within Idle (**File > New File**) then choose **File > Save As** and save your file as Ex4.py in the C:\Labs\Exercise 4\Starter directory.
2.	Declare a variable called count and assign zero to it `count = 0`

3.	Add 5 to the count
	`count = count + 5`
	or
	`count += 5`

4.	Multiply count by 2

	`count = count * 2`
	or
	`count *= 2`

5.	Print the value and the type of the count variable `print(count, type(count))` From the Run Menu choose Run Module (you will need to save your script)
6.	Multiply count by exactly 1.0 `count *= 1.0`
7.	Print the value and the type of the count variable again. From the Run Menu choose Run Module (you will need to save your script) What's changed? Can you explain it?

Part 2 – Performing conversions

4.	Using the following data
	`title = "Jaws"`
	`duration = 91.0`
	`year = 1994`
	Create a variable called **summary** that holds a summary of the movie details including all of the above data. When you print it out, it should read something like this:
	My favourite movie is Jaws. It was filmed in 1994 and runs for 91 mins
	The problem you face is strings and numbers don't mix with the + operator. You will need to convert one or more variables to a string before adding it to another string.
	In Python you can convert numbers to string using **str()** and strings to numbers using **int()** and **float()**. A simple example of this would be.
	`x = 5`
	`print("x is " + str(x))`

	Try this yourself and if you are not sure have a look at the answer section below.

Part 3 – Performing calculations

5.	Declare variables for a vehicle travelling with the following parameters 1. Speed: 47 mph 2. Time: 2.5 hours 3. Distance?
6.	Write code to determine how far the vehicle will have travelled and print out the result. **Hint: Multiply Speed by Time to get distance**

(If you have time)

Using the **input** function to capture user input, ask the user to enter the following data:

- Loan Amount
- Interest Rate

Calculate the simple interest to be paid on the loan for the interest rate in one year. Either work out how to do this by yourself or you can follow these steps:

Steps

7.	Capture the loan amount and the interest rate:
	`loan_text = input("Enter Loan Amount: ")`
	`rate_text = input("Enter Interest Rate: ")`

8.	Convert the text variables to float variables:
	`loan = float(loan_text)`
	`rate = float(rate_text)`

9.	Perform the calculation and print the result:
	`interest = loan * rate / 100`
	`print("Interest is", interest)`

Answers

Part 1

```
count = 0
count += 5
count *= 2
print(count, type(count))
count *= 1.0
print(count, type(count))
```

Part 2

```
title = "Jaws"
duration = 91.0
year = 1994
summary = "My favourite movie is " + title + ". It was filmed in
" + str(year) + " and runs for " + str(duration) + " mins "
print(summary)
```

Part 3

```
speed = 47
time = 2.5
distance = 47 * 2.5
print("Distance travelled " + str(distance))
```

Part 4

```
loan_text = input("Enter Loan Amount: ")
rate_text = input("Enter Interest Rate: ")
loan = float(loan_text)
```

```
rate = float(rate_text)
interest = loan * rate / 100
print("Interest is ", interest)
```

Exercise 5 – Collections

Objective

In this exercise we will use collections to organize data into related containers of information.

Duration

15-20 minutes.

Overview

Most people have more than one favourite movie; in fact, we all probably have at least a top 5. The desire to group and classify data into a category is a common human behaviour and one we use in computing as well.

In this exercise, we will introduce the idea of lists and dictionaries and how we might use them to organise our movie data into a coherent set of information.

Part 1 – Creating a top 5 list

To define a list of people you know, you might write code similar to:

```
people = ["Fred", "Amy", "Tam", "Suzy", "Bob"]
```

Notice the use of square brackets to define the list. Any comma-separated list of items inside the [and] become part of the list.

To print the first person in the list we would write:

```
print(people[0])    # item operator after a sequence gets index
```

Your task is:

Using your IDE, create a new script in C:\Labs\Exercise 5\Starter called **topFive.py** then:

1.	Create a list of your favourite movie titles and assign it to a variable called topFive. `topFive = ["Star Wars", "Jaws", "Postman Pat 2", "Vanilla Sky", "Absolutely Fabulous"]`

2.	Print out your list. `print(topFive)`
3.	Print out just the title of your second favourite movie (no help but you could look at the Answers below). **Hint**: First item is at position 0
4.	Add one more movie to your list after you have defined it and then print the list. **Hint:** You are trying to append items to the list
5.	Print the number of items in your list. **Hint: len()** function may help
6.	Change the last item in your list to a different movie title and print out the list again. The last item in the list should have changed.

Now, although this doesn't make sense, why not finish off by sorting your top five list. Use the **sorted** function to do it. Notice when you sort your list you don't change the actual list, you create a new sorted list.

Part 2 – Using Dictionaries to store key:value pairs (if you have time)

Dictionaries can be used to store unique key and value pairs of data.

```
person = {"firstName": "Fred", "lastName": "Bloggs"}
```

The : binds the key to the value.

The key must be unique and can be used to retrieve the value from the dictionary in the same way you retrieve the meaning of a word from a dictionary.

```
print(person["lastName"])
```

7.	We would now like you to create a dictionary to hold your favourite movie info containing details such as: • title • year • director • duration

	To find info for your movie, take a look at http://www.imdb.com. It might start out looking something like this. You need to finish yours off. `movie = {"title": "Jaws", "director": "Stephen Spielberg"}`
8.	Using your dictionary print the title, year and director. To print the director in the dictionary above we would write: `print("Director:" + movie["director"])`

Part 3 (If you have time)

Think about how you might create a list of movies with each movie held in its own dictionary. Write the code for it if you feel confident (ask the instructor for help). The solution is described below.

Answers

Part 1

```
topFive =[ "Star Wars", "Jaws", "Postman Pat 2", "Vanilla Sky",
"Absolutely Fabulous"]

print(topFive)

topFive.append("One Flew over the Cuckoos nest")

print(topFive)

print(len(topFive))

topFive[-1] = "It's a beautiful life"

print(topFive)
```

Part 2

```
movie = {"title": "Jaws", "director": "Stephen Spielberg",
"year": 1975, "duration": 124.0}

print("Title:" + movie["title"])

print("Year:" + str(movie["year"]))

print("Director:" + movie["director"])
```

Part 3

```
movies = [{"title": "Jaws", "director": "Stephen Spielberg",
"year": 1975, "duration": 124.0},

{"title": "Star Wars", "director": "Stephen Spielberg", "year":
1977, "duration": 91.0}]

print("Title:" + movies[1]["title"])

print("Year:" + str(movies[1]["year"]))

print("Director:" + movies[1]["director"])
```

Exercise 6 – Control Flow

Objective

In this exercise, we will control the flow of execution of code using if statements and loops.

Duration

20-25 minutes.

Overview

Few programs execute as a straight sequence of instructions and many use logical tests to control the flow of execution. A program's state is ever changing and the program may need to respond by executing different sections of code conditionally.

To achieve control, we use structures like if statements and while loops to manage decisions and repetition within the code.

Part 1 – Selectively processing a List

In Python, an if statement looks like this:

```
if x > y:
    print "x is greater than y"
else:
    print "The alternative"
```

1.	In your IDE, load the script MovieArray.py in folder C:\Labs\Exercise 6\Starter. Notice there is a rather eclectic list of movies defined and a line of code that allows the user to enter a new movie title.

Let's add some code to test if the list is empty. If it isn't, print a message describing the length of the list. Otherwise print a message that just says "Empty List".

Steps

2.	Add an if clause to test the length of the list.
	`if len(movies) > 0:`
	`    print(len(movies), "movies in list")`

3.	Now add an else statement to cater for the empty list.
	`else:`
	`    print("Empty list")`

We now want you to write some code that adds the newly entered movie title **new_movie** to the movie list **but** only if it doesn't already exist. Then print the modified movie list.

To check if a list contains an item you could use syntax similar to:

```
numbers = [87, 45, 65, 32, 45]
if 65 in numbers:
    print("65 exists")

else:
     numbers.append(65)
```

Either try to do this yourself or you could follow these steps:

Steps

4.	Add an if statement to check if the entered movie is in the list and print a message if it is.
	`if new_movie in movies:`
	`    print("Movie Exists")`

5.	Then add an else statement to append the new movie to the list.
	`else:`
	`    movies.append(new_movie)`
	`    print(movies)`

Part 2 – Iterating through a List

6.	In your IDE, load the script MovieLoop.py in folder C:\Labs\Exercise 6\Starter.
7.	Write a loop to iterate through the movies list and print out each movie one at a time.

Either do this yourself or follow these steps:

Steps

8.	Add a for loop that iterates through each item in the list. `for movie in movies:`
9.	Each movie in the list will be held in the movie variable defined inside the loop (you can call this variable anything you like). Inside the loop add code to print each item `    print(movie)`

	The final result would be:
	`for movie in movies:`
	`    print(movie)`

Part 3 – Iterating through a list of dictionaries (If you have time)

10.	In your IDE, load the script MovieDictionary.py in folder C:\Labs\Exercise 6\Starter.
11.	Write a loop to iterate through the list of dictionaries one at a time. Each dictionary contains movie info. Print out the title and director of each movie.

Part 4 – Search the dictionary (if you have time)

12.	In your IDE, load the script MovieSearch.py in folder C:\Labs\Exercise 6\Starter.
13.	The code allows the user to enter the title of a movie. Use the entered title to search the list of movie data for a matching title. If there is a match, print out the movie details. To do this: Write a loop (either a for loop or while loop) that: • iterates through the list of movie data (as above); • finds a movie with the entered title. If you find a matching movie, then print out its details. • finally, alter the code to indicate if the requested movie was not found (This may make use of *break* and the *else* clause on the *for* loop) Ask your instructor about these steps if in doubt

Answers

Part 1

```
movies = ["Jaws", "Star Wars", "Vanilla Sky", "The Mission",
"Bridget Jones Diary 2"]

print(movies)

new_movie = input("Enter Movie Title to Add: ")

if len(movies) > 0:

    print(len(movies), "movies in list")

else:

    print("Empty list")


# Check if movie exists, if it doesn't then append to array,
otherwise say it exists

if new_movie in movies:

    print("Movie exists!")

else:

    movies.append(new_movie)

    print(movies)
```

Part 2

```
movies = ["Jaws", "Star Wars", "Vanilla Sky", "The Mission",
"Bridget Jones Diary 2"]


# Iterate through each movie and print out

for movieTitle in movies:

    print(movieTitle)
```

Part 3

```
movies = [{"title": "Jaws", "director": "Steven Spielberg",
"year": "1975"},

{"title": "Star Wars", "director": "Steven Spielberg", "year":
"1977"}, {"title": "Vanilla Sky", "director": "Cameron Crowe",
"year": "2001"},

{"title": "The Mission", "director": "Rolan Joffe", "year":
"1986"},

 {"title": "Bridget Jones Diary", "director": "Sharron McGuire",
"year": "2001"}]

# Iterate through each movie and print out title

for movie in movies:

    print("Title:", movie["title"])

    print("Director:", movie["director"])
```

Part 4

```
movies = [{"title": "Jaws", "director": "Steven Spielberg",
"year": "1975"},

{"title": "Star Wars", "director": "Steven Spielberg", "year":
"1977"},

{"title": "Vanilla Sky", "director": "Cameron Crowe", "year":
"2001"},

{"title": "The Mission", "director": "Rolan Joffe", "year":
"1986"},


{"title": "Bridget Jones Diary", "director": "Sharron McGuire",
"year": "2001"}]

title = input("Enter Movie Title: ")

for movie in movies:

    if movie["title"] == title:

        print("Title:" + movie["title"])

        print("Director:" + movie["director"])

        print("Year:" + movie["year"])
```

Part 4a

```
movies = [{"title": "Jaws", "director": "Steven Spielberg",
"year": "1975"},

{"title": "Star Wars", "director": "Steven Spielberg", "year":
"1977"},

{"title": "Vanilla Sky", "director": "Cameron Crowe", "year":
"2001"},

{"title": "The Mission", "director": "Rolan Joffe", "year":
"1986"},

{"title": "Bridget Jones Diary", "director": "Sharron McGuire",
"year": "2001"}]

title = input("Enter Movie Title: ")

for movie in movies:

    if movie["title"] == title:

        print("Title:" + movie["title"])

        print("Director:" + movie["director"])

        print("Year:" + movie["year"])

        break      # This jumps over the else:

# The else belongs to the for loop, not the if statement

else:

    print(title, "not found")
```

Exercise 7 – Structuring your Program

Objective

It's now time to add some structure to our programs. In the process, we will learn how to use functions to achieve this goal.

Duration

20-30 minutes.

Overview

It's often said as developers, we stand on the shoulders of giants. That usually means we reuse functionality written by others.

Part 1 – Reusing existing modules and libraries

You can reuse all of the functionality defined within the modules and libraries that are part of the Python language. In addition, you can also define your own modules and functions.

To reuse existing modules, you just need to import the ones required (as long as they aren't already built in).

So, let's reuse some functionality.

1.	First let's generate a random number. To do that we need to import the **random** module into our script. In your IDE, create a new file called **part1.py** and enter the following import statement: `import random`

2.	Next, to access functions within this module, we need to use the name of the module first and then access the **randint** function contained inside.
	`random_num = random.randint(1, 49)`
	`print(random_num)`
	This will generate a random number between 1 and 49.
	Every time we run this code it should produce a different number.

We just reused an existing function contained within a pre-defined module.

Part 2 – Making your code reusable with functions

When you write your own code, you will likely want to make at least some of it reusable by other parts of your application and even for other applications.

To make the following code reusable, we might wrap it up inside a function:

```
amount = 5000

rate = 4.5

interest = amount * rate / 100

print(interest)
```

The function equivalent would look like this:

```
def calc_interest(amount, rate):
     interest = amount * rate / 100
     return interest
print(calc_interest(15000, 3.5))
```

Now take a look at the following code and then open it in your IDE inside the days.py script in the C:\Labs\Exercise 7\Starter folder:

```
day = "mon"

cap_day = day.capitalize()

full_day = cap_day + "day"

print(full_day)
```

The purpose of the script is to take a short day string and convert it into a full day string. That means "sun" would become "Sunday", etc.

How could you make this code reusable (without copy paste)? Your task is to turn this code into a function called **expand_day** that takes a day parameter and returns a fully expanded day string.

Either work out how to do this yourself or you can follow these steps:

Steps

3.	Define a function called expand_day with a day parameter. `def expand_day(day):`

4.	Copy lines 2 and 3 of your original code into the function and indent each line with a tab.
	`def expand_day(day):`
	`    cap_day = day.capitalize()`
	`    full_day = cap_day + "day"`

5.	Return the **full_day** local variable from the function.
	`def expand_day(day):`
	`    cap_day = day.capitalize()`
	`    full_day = cap_day + "day"`
	`    return full_day`

6.	Invoke your function by calling it with a supplied parameter value. `print(expand_day("mon"))`

Part 3 – Movie search (if you have time)

Open C:\Labs\Exercise 7\Starter\movielib.py script in your IDE. It contains a list of movie dictionaries. Each movie dictionary contains a title, year, director, etc.

If you run the script then you will be able to use it to search for movie titles in the data.

We want to convert this script so that it contains a reusable function called **find_movie** that takes a title parameter and returns a movie dictionary when it finds one that matches the title.

You may want to figure out how to do that on your own but you could also follow the steps below.

Steps

7.	Remove the line that captures the movie title using the input function. ~~`title = input("Enter Movie Title:")`~~
8.	Create a function (above line **found = None**) called **find_movie** that takes a title parameter `def find_movie(title):`

9.	Highlight and indent all the code below to create the following:
	`def find_movie(title):`
	`    found = None`
	`    count = 0`
	`    while count < len(movies) and found is None:`
	`        movie = movies[count]`
	`        if movie["title"] == title:`
	`            found = movie`
	`        else:`
	`            count += 1`
	
	`    if found is not None:`
	`        print("Title:" + found["title"])`
	`        print("Director:" + found["director"])`
	`        print("Year:" + found["year"])`

10.	Remove the if statement and the associated prints.

	~~if found != None:~~ ~~print("Title:" + found["title"])~~ ~~print("Director:" + found["director"])~~ ~~print("Year:" + found["year"])~~
11.	Replace it with a return statement that returns the found variable. This found variable is either the found movie dictionary or None, which means not found.
12.	Save your module.

Now to test it, you need to open C:\Labs\Exercise 7\Starter\moviesearch.py in your IDE:

13.	Notice this script imports the **movielib** module.

14.	Add a call to the find_movie function inside the movielib module by adding the following line:
	`movie = movielib.find_movie("Jaws")`
	`print(movie)`

Answers

Part 2

```
def expand_day(day):


    cap_day = day.capitalize()


    full_day = cap_day + "day"


    return full_day



print(expand_day("sun"))
```

Part 3

Movielib.py

```
movies=[{"title": "Jaws", "director": "Steven Spielberg",
"year": "1975"},

 {"title": "Star Wars", "director": "Steven Spielberg", "year":
"1977"},

{"title": "Vanilla Sky", "director": "Cameron Crowe", "year":
"2001"},

{"title": "The Mission", "director": "Rolan Joffe", "year":
"1986"},

{"title": "Bridget Jones Diary", "director": "Sharron McGuire",
"year": "2001"}]

def find_movie(title):

    found = None

    count = 0

    while count < len(movies) and found is None:
```

```
        movie = movies[count]
        if movie["title"] == title:
            found = movie
        else:
            count += 1

    return found
```

Moviesearch.py

```
import movielib
movie = movielib.find_movie("Jaws")

print(movie)
```

Exercise 8 – Controlling Data Scope

Objective

Understanding data scope is an important part of programming. In this exercise, we will examine how to control the scope of data variables we create.

Duration

15-20 minutes.

Overview

You will be writing code to control the scope of data used in your programs.

Part 1 – Accessing global variables

Open C:\Labs\Exercise 8\Starter\scope.py and have a look at the code within.

```
name = "Fred"
def display_details():
    age = 21
    print("Name:", name)
    print("Age:", age)
```

The name variable is global to the script but age is defined inside the display_details function.

1.	Try running the script. Why does nothing happen?
2.	Find the comment **#Part 1: Print Name** and add some code below to print out the name variable. `print(name)`
3.	Run the script again. Does it print the name?

The reason why name prints is because it is a global variable to the script.

4.	At the bottom of the file, add the following code: **print(globals())** You will see a list of global variables known to Python including **name** and the **display_details** function.

Part 2 – Accessing local variables

Take a look at the **display_details** function. Inside we are defining a variable called age. Let's try printing age.

5.	Locate comment #Part 2: Print Age?
6.	print the age variable. `print(age)` Will this work?
7.	Run the script.

Age is defined within the function and only exists within that function while the function is executing. This means you can isolate data inside a function making it inaccessible by the caller of the function.

How could you make age accessible to the caller?

Part 3 – Call display_details

display_details will only execute if we call it by adding parentheses at the end of its name.

```
display_details()
```

8.	Comment out the print age line to stop it erroring. `#print(age)`
9.	Locate comment #Part 3: Call display_details.
10.	Add code to execute the display_details function then run the script. Will it work?

The result prints both the name variable and the local age variable. That's because functions have access to variables declared at the script level (global) and also any variables that are declared locally.

Questions

How might you change display_details so that it is no longer reliant on a global variable? Why not make the change?

How would you make the age data accessible outside of the function?

Part 4 – Object scope (if you have time)

Open scripts movie.py and showmovies.py.

11.	Take a look at movie.py. In it, we have defined some variables for a movie's title, director and year. There is also a function that accesses the global variables and prints them.
12.	Look at showmovies.py. It imports the movie.py module and then calls the display_details function.
13.	In showmovies.py, below the display_details() call, add code to change the movie title, director and year to something else. Then call display_details again. You will need to prefix all access with the module name **movie**.

To represent different movies, we need to change the same variables each time.

Ideally, we want to organize the movie data into a container that we can use to store movies separately in memory. To do that, we need to define a class.

To define a class for a Person we might write something like:

```
class Person:
     def __init__(self):
          self.first_name = ""
          self.last_name = ""
          self.age = 0
```

To then create a Person object, we would write:

```
person_1 = Person()
person_1.first_name = "Fred"
```

```
person_1.last_name = "Bloggs"
```

We can now create multiple instances of Person objects each with their own separate identity in memory that hold their own distinct data values for each of the fields.

Let's use these same principles to convert the movies.py module into a class implementation. You can do this in one of two ways, either by doing it yourself from the requirements described below or by following the described steps that are also provided.

Requirements

14.	In movies.py we want you to convert the code into a class implementation. Write a class called Movie that will hold the following data: • title • director • year Add the display_details function to it.
15.	In showmovies.py, create multiple instances of movies and assign each movie its own distinct data values for title, director and year. Call display_details on each one.

Steps

16.	At the top of movie.py add a new line: `class Movie:`

17.	Add a new function called **def __init__ (self):**
	That's two underscores at the start and two underscores at the end.
	And indent/modify the lines that create the title, director and year variables to give:
	`class Movie:`
	`    def __init__(self):`
	`        title = ""`

```
        director = ""
        year = 0
```

18. Prefix each variable with **self**.

```
class Movie:
    def __init__(self):
        self.title = ""
        self.director = ""
        self.year = None
```

19. Now indent the display_details function and its print statements to produce:

```
class Movie:
    def __init__(self):
        self.title = ""
        self.director = ""
        self.year = None

    def display_details():
        print("Title:", title)
        print("Director:", director)
        print("Year:", year)
```

20. To enable objects to access the data fields, we need a reference to the object. Add a **self** parameter to **display_details** and prefix each of the title, director and year fields with **self**.

```
def display_details(self):
    print("Title:", self.title)
    print("Director:", self.director)
    print("Year:", self.year)
```

21.	Open showmovies.py, comment out the existing code (except the import statement(s)) and create an instance of a movie. `first_movie = movie.Movie()`

22.	Fill in some movie details for the object you just created.
	`first_movie.title = "Jaws"`
	`first_movie.director = "Steven Spielberg"`
	`first_movie.year = 1975`

23.	Call display_details on this movie. `first_movie.display_details()`
24.	Create another movie with different details in a variable called second_movie.

Part 5 – (If you really, really have time)

25.	Change the **__init__(self)** function to have three parameters
	• title
	• director
	• year
	Assign each parameter value to its corresponding class attribute.
	`def __init__(self, title, director, year):`
	`    self.title = title`
	`    self.director = director`
	`    self.year = year`
	When you create a Movie you can now pass the movie details in as initialisation arguments.
	`first_movie = movie.Movie("Jaws", "S. Spielberg", 1975)`

Part 6 – (If you really, really, really have time)

26.	Create a Dictionary of movies and use the IMDB ID as the unique key (IMDB ID is the identifier used in the URL for a Movie entry in the IMDB website; it is prefixed with **tt**).

Exercise 9 – Accessing and persisting data

Objective

To understand how to persist data stored in memory to a persistent storage location.

Duration

15-25 minutes.

Overview

When a program finishes executing or the computer it was running on switches off, any memory used to store data is lost. That means your data is lost. We need a way of saving data beyond the lifespan of a single program execution.

There are lots of ways to do this, including local and remote database storage as well as many cloud storage options. We will look at file storage in this exercise.

You will be working in folder C:\Labs\Exercise 9\Starter that contains:

- movie.py
- movies.txt

Part 1 – Reading data from a file

1.	Open movies.txt by double clicking on it. Notice it contains info about movies.
2.	Create a script called fileprocess.py in C:\Labs\Exercise 9\Starter.
3.	Open the movies.txt file using the following code: `data_file = open("movies.txt", "r")`
4.	Next let's write a for loop to retrieve each line of data from the file using the file reference returned above: `for line in data_file:`
5.	Now inside the loop, print the retrieved line variable (you could call this variable anything, it doesn't have to be line): `for line in data_file:`

	`print(line)`

6.	Now close the file as highlighted in bold.
	`for line in data_file:`
	`  print(line)`
	`data_file.close()`

7.	Run your script and you should print out the contents of the file. Why is there a blank line between each movie?

Part 2 – Writing data to a file

Did you notice how all of the text data is in lower case? We could capitalise the data line read in using the **capitalize** function but that would only capitalise the first word. To capitalise all words, we use the **title** function.

Let's convert the input file into an output file called **movies_out.txt** that contains capitalised versions of each line of data.

Either try to do this yourself or follow these steps:

Steps

8.	After the first **open** statement add a new open statement on a new line to open the movies_out.txt file for writing: `data_file_out = open("movies_out.txt", "w")`

9.	Now add a file write statement to the for loop to write the capitalised data line to the output file:
	`for line in data_file:`
	`  print(line.title())`
	`  data_file_out.write(line.title())`
	or better would be:
	`for line in data_file:`
	`  processed_line = line.title()`
	`  print(processed_line)`

	`data_file_out.write(processed_line)` Why is that better?
10.	Finally close the output file. **`data_file_out.close()`**
11.	Run your script and you should write out the capitalised contents of the input file to the output.

Part 3 – Capturing data and storing it in a file (if you have time)

Let's combine the data capture skills obtained earlier on in the course using the **input** function and ask for a user's favourite movie title, director and year.

This time though we should store the data in a file called **topFive.txt**. The twist is the script should append to the file every time it runs.

Either try to do this yourself or follow these steps.

You should create a module called moviecapture.py.

Steps

12.	Using **input,** record the title, director and year.
	`title = input("Enter Movie Title: ")`
	`director = input("Enter the Director: ")`
	`year = input("Enter Year of Release: ")`

12.	Open the file topFive.txt for append: `data_file_out = open("topFive.txt", "a")`
13.	Create a variable to hold the output data and assign the title, director and year joined together with commas "," between them. `movie_data = title + "," + director + "," + year`
14.	Add a new line character to the end of your movie data string so each movie is separated from each other in the file.

	movie_data += "\n"

15.	Write the movie data to the output file and close the output file:
	`data_file_out.write(movie_data)`
	`data_file_out.close()`

16.	Run the script, enter movie data and check you produce a topFive.txt file with data in it. Rerun to make sure you are appending data to the file.

Part 4 – Importing and processing structured data (if you have time)

What about processing comma-separated data stored in a file? Once you have read comma-separated values in as a string you will probably want to extract the data from the string into individual variables.

We can convert the following code in C:\Labs\Exercise 9\Starter\movieprocess.py to process the **line** variable within the for loop and extract the title, director and year data from it.

```
import movie


data_file = open("movies_out.txt", "r")
movies = []
for line in data_file:
    print line


data_file.close()
```

Python strings have a **split** function that allow components of a string to be separated from each other based on a separator character. For comma-separated, the separator is a comma so we could use the split function like this:

```
movie_data = line.split(",")
```

That returns a list of the values between the commas.

Jaws, Steven Spielberg,1975

This would split to a list:

[0] Jaws
[1] Steven Spielberg
[2] 1975

We could then create an instance of a **movie.Movie** and set each movie's title, director and year fields to each value in the list.

Once you have created and filled in a movie, append it to the movies list.

At the end, write another for loop to iterate through the movies list printing each movie's details.

Try to work this one out yourself and have a look at the Answers section if you need hints.

Answers

Part1

```
data_file = open("movies.txt", "r")

for line in data_file:
    print(line.title())

data_file.close()
```

Part 2

```
data_file = open("movies.txt", "r")
data_file_out = open("movies_out.txt", "w")

for line in data_file:
    print(line.title())
    data_file_out.write(line.title())

data_file_out.close()
data_file.close()
```

Part 3

```
title = input("Enter Movie Title: ");
director = input("Enter the Director: ")
year = input("Enter Year of Release: ")

data_file_out = open("topFive.txt", "a")
movie_data = title + "," + director + "," + year
movie_data += "\n"
data_file_out.write(movie_data)
data_file_out.close()
```

Part 4

```
import movie


data_file = open("movies_out.txt", "r")
movies = []
for line in data_file:
    items = line.split(",")
    new_movie = movie.Movie()
    new_movie.title = items[0]
    new_movie.director = items[1]
    new_movie.year = items[2]
    movies.append(new_movie)

data_file.close()

for current in movies:
    print(current.title)
```

Exercise 10 – Building and Testing Applications

Objective

To understand the basic principle of building and testing applications.

Duration

15-25 minutes.

Overview

Part 1 – Building and running applications

Scenario	Compiled	Interpreted	Runtime
You can build my source code into an executable My executable contains machine instructions			
You can build my source code into an executable The executable contains byte-code not machine instructions Executable cannot be executed directly by the OS and requires additional software installed on the OS			
My source code is executed dynamically a line at a time			

What might be some of the advantages and disadvantages of each approach?

Part 2 – Testing

One of the best habits you can get into when writing code is to write tests for the functions you develop. Your tests can be used to prove your function works the ways it's supposed to.

We have written a new module called **movie_library.py**. Open it in your IDE from the C:\Labs\Exercise 10\Starter folder.

We have also written a module called **test_movie_library.py**. Open that as well.

1.	Open test_movie_library.py and find the first comment. # Add a new movie. Add some code to add a new movie to the movie library. `library.add("Star Wars", "Steven Spielberg", 1977)`
2.	Now add an assertion check to test if your movie has been added. You can use the library.count() function to determine the movie count. It should be one. `self.assertEqual(library.count(), 1)`
3.	Run the script. It should report the number of tests and OK.

We can now write another test to check that we can remove movies based on their position.

4.	Inside test_remove_movie add a call to remove the first movie in the list (position 0). `library.remove(0)`
5.	Now add an assertion check to test if your movie has been removed. It should be zero. `self.assertEqual(library.count(), 0)`
6.	Run the script. It should report the number of tests and OK.

Part 3 – Write more tests (if you have time)

Take a look at movie_library.py. What additional tests would you write to make the testing more robust?

You could try changing the code to prevent adding duplicate movies and then write a test to ensure it works. Even better, write the test first then change movie_library.py to implement the prevention of duplicates.

Answers

Part 1

Scenario	Compiled	Interpreted	Runtime
You can build my source code into an executable My executable contains machine instructions	•		
You can build my source code into an executable The executable contains byte-code not machine instructions Executable cannot be executed directly by the OS and requires additional software installed on the OS			•
My source code is executed dynamically a line at a time		•	

Part 2

```
from movie_library import *
import unittest
class TestMovie(unittest.TestCase):
    def test_add_movie(self):
        library = MovieLibrary()
        library.add("Star Wars", "Steven Spielberg", 1977)
        self.assertEqual(library.count(), 1)
    def test_remove_movie(self):
        library = MovieLibrary()
        library.add("Star Wars","Steven Spielberg", 1977)
        self.assertEqual(library.count(), 1)
```

```
        library.remove(0)
        self.assertEqual(library.count(), 0)
    def test_find(self):
        library = MovieLibrary()
        library.add("Star Wars","Steven Spielberg", 1977)

        found_movie = library.find_by_title("Star Wars")
        self.assertNotEqual(found_movie, None)

if __name__ == '__main__':
    unittest.main()
```

Exercise 11 – Coding style (Optional)

Objective

To consolidate the ideas in this chapter we would like your opinion on some code we have written. You do not need to understand what the code does (we made it up) but to look at the *style*, and decide which is better.

Questions

1. Which of the following code fragments is likely to run faster?

   Which is easier to read, and which will be easier to debug?

   The code in a) and b) do the same thing (are *functionally equal*).

   a) All in one line:

```
display(get_item(get_file(i)) for i in get_envs())
```

   This is an example of *nested function calls*, where the returned value of one function is used as the parameter that is passed to another. Putting the `for` loop at the right-hand side is known as a *statement modifier*.

   b) Written over several lines using several intermediate variables:

```
env_list = get_envs()
```

```
for env in env_list:
    file = get_file(env)
    item = get_item(file)
    display(item)
```

2. Is there anything wrong with the following variable names?

```
a
iCount
variable
if
extraopolationofcoordinates
2pair
Countl
iT6ToTaL
EXIT
```

Answers

1. Which of the following code fragments is likely to run faster?

   There will probably be no difference in the run-time of these code fragments.

   a) All in one line:

```
display(get_item(get_file(i)) for i in get_envs())
```

   The first fragment a) will require the compiler to use several temporary intermediate values, the return values from each nested function call. It is difficult to read, partly because the eye has to dart around from left-to-right and then right-to-left. It is difficult to debug because you cannot see what the temporary values are from each function call, you cannot insert print statements anywhere, and, in a debugger, you will have difficulty in stepping into one particular function when required.

   In its defence, many practitioners would consider this code to be elegant, in that it is concise in form.

   b) Written over several lines using several intermediate variables:

```
env_list = get_envs()


for env in env_list:
    file = get_file(env)
    item = get_item(file)
    display(item)
```

   The second fragment b) is verbose but clearer. It progresses in simple "top-down" steps. It is easy to debug because we can inspect each value as it is returned by each function call. In a debugger, we can easily pick which function to step into if necessary.

   A disadvantage is that it uses variables like file *and* item *which are only necessary for readability, so you might think these could slow things down. That is unlikely. Most modern compilers include optimisation which will remove cosmetic variables*

and produce a fast program. But you can't expect an optimiser to rewrite your program and get rid of mistakes.

The verbosity of this fragment is its downfall; it is difficult to see what it is doing at a glance: you have to step though each line.

So, which is better? In performance, there is nothing in it. For readability and style, it is in the eye of the beholder: Most beginners would consider b) to be easier, but experienced programmers might not see anything wrong with a). For debugging, b) is the clear favourite.

2. Is there anything wrong with the following variable names?

`a`

The name is not descriptive, usually. There are some algorithms where a single 'a' would be acceptable, such as a sort, but these are rare.

`iCount`

Most would say there is nothing wrong with this, it is perfectly acceptable. However it uses a convention known as Hungarian Notation that some (e.g. Google) do not like.

`variable`

The name is meaningless; it does not carry any information.

`if`

This is a keyword in most programming languages, and would be illegal.

`extraopolationofcoordinates`

The length of this variable, the complication of the words and the lack of word boundaries all make this a bad choice. It is inviting a mistake to be made.

`2pair`

This name starts with a number, which is illegal in most programming languages.

```
Countl
```

The final letter l (ell) of this name could be misinterpreted as a 1.

```
iT6ToTaL
```

Too complicated! The poor user of this name has to continually shift in and out of upper and lower case. This is another example of an invitation to make a mistake. Does it mean i-T-6-Total or iT6-To-Tal? TAL is a language used on Tandem Non-Stop systems.

```
EXIT
```

Uppercase variables are often discouraged and used to indicate system variables, constants or labels.

Appendix 1 – Extra grid sheets